Evictor

Prashant Jain

Prashant.Jain@mchp.siemens.de

Siemens AG, Corporate Technology

Munich, Germany

Evictor

The Evictor[1] pattern describes how and when to release resources such as memory and file handles to optimize resource management.

Example Consider a network management system (NMS) that is responsible for managing several network elements (NEs). These NEs are typically represented in a topology tree. A topology tree provides a virtual hierarchical representation of the key elements of the network infrastructure. The NMS allows a user to view such a tree as well as get details about one or more NEs. Depending upon the type of the NE, its details may correspond to a large amount of data. For example, the details of a complex NE may include information about its state as well as the state of its components.

The topology tree can be constructed at application start-up or when the user asks to view the network of NEs, or some time in between. Similarly, the details of all the NEs can be fetched as the tree is constructed or can be deferred until the user makes a request for it. However, regardless of when the details of the NEs are brought into memory, keeping these details in memory can be quite expensive. If the details of an NE are never accessed by the user again, they will consume valuable resources in the form of memory. On the other hand, the user may request the details of the same NEs and therefore keeping the details in memory (a.k.a caching) can be desirable to improve performance. If the details of an NE that is frequently accessed by a user are not cached, it can lead to expensive calls being made to the real NE to get its details. This in turn can degrade system performance.

Context Systems that need efficient management of resources.

Problem Highly robust and scalable systems must manage resources efficiently. A resource can include local as well as distributed objects and services. Over time, an application acquires many resources some of which are only used once. If an application keeps on acquiring resources without ever releasing them, it will lead to performance degradation along with system instability. To avoid this problem, the application may immediately release resources after using them. But the application may need to use the same resources again, which would require re-acquisition of those resources. However, re-acquisition of resources can itself be expensive and should therefore be avoided by keeping frequently used resources in memory. To address these conflicting requirements of resource management requires the resolution of the following forces:

- The frequency of use of a resource should influence the lifecycle of a resource.
- Resource release should be determined by parameters such as type of resource, available memory and CPU load.
- The solution should be transparent to the user.

Solution Monitor the use of a resource and control its lifecycle using some form of strategy such as Least Recently Used (LRU) or Least Frequently Used (LFU). Each time a resource is used, it is marked by the application. A resource that is not recently used or not frequently used does not get marked. Periodically or on demand, the application selects the resources that are not marked and releases or evicts them. Resources that are marked continue to stay in memory since they are used frequently.

1. The idea behind the Evictor Pattern was first described in [HeVi99].

Alternatively, other strategies can be used to determine which resources to evict. For example, for memory-constrained applications, the size of resources can be used to determine which resource(s) to evict. In such case, a high-memory consuming resource may get evicted even if it was recently used.

Note that the solution only focuses on strategies for resource release/removal. Solutions for resource acquisition are described in Lazy Acquisition [Kirch01], Leasing [JaKi00], and Lookup [KiJa00] patterns.

Structure The following participants form the structure of the Evictor pattern:

A *resource* provides some type of functionality or service and includes local as well as distributed objects and services

A *user* uses a resource and can include an application or an operating system.

An *evictor* evicts resources based on one or more eviction strategies.

An *eviction strategy* describes the criteria that should be used to determine if a resource should be evicted or not.

The following CRC[2] cards describe the responsibilities and collaborations of the participants.

Class User	***Collaborator***
Responsibility	• Resource
• Uses a resource	

Class Resource	***Collaborator***
Responsibility	• User
• Provides application functionality	

Class Evictor	***Collaborator***
Responsibility	• Resource
• Evicts resources based on one or more eviction strategies	• Eviction Strategy

Class Eviction Strategy	***Collaborator***
Responsibility	
• Describes criteria to use to determine which resource to evict.	

Implementation There are four steps involved in implementing the Evictor pattern.

1 *Define eviction interface*: An eviction interface should be defined that will be implemented by all resources that can be evicted. For example, the eviction interface in Java may look like:

```
public interface EvictionInterface {
     public boolean isEvictable ();
     public Object info ();
     public void beforeEviction ();
}
```

2. Class-Responsibility-Collaborators (CRC) cards help to identify and specify objects or the components of an application in an informal way, especially in the early phases of software development. A CRC-card describes a component, an object, or a class of objects. The card consists of three fields that describe the name of the component, its responsibilities, and the names of other collaborating components.

The method `isEvictable()` can be used to determine if an object is evictable. Please see step 2 for details. The method `info()` is used by the Evictor to extract strategy-specific information from the object to determine whether or not to evict it. Please see step 4 for details. The method `beforeEviction()` serves as a hook method that can be called by the Evictor before it evicts an object. This gives the object a chance to release any resources it may have acquired.

> For example, the EJB Session Bean and Entity Bean interfaces include a method called `ejbPassivate()` that is called just before an entity or a session bean is evicted. This gives the bean a chance to release any acquired resources.

2 *Determine evictable resources*: The developer must determine which resources can and should be evicted. For example, resources that are always required by an application or those that can not be re-acquired should not be evicted. Any resource that can be evicted must implement the eviction interface. Prior to evicting the resource, the application should call the interface giving the resource a chance to do any necessary clean-up including persisting any necessary state.

In the Java interface described above, the application can use the method `isEvictable()` to indicate if a resource can be evicted. If the method returns `true`, the resource is considered by the Evictor as a possible candidate for eviction; if the method returns `false`, the Evictor ignores the resource.

3 *Determine eviction strategy*: Based on application requirements, different eviction strategies can be used in determining when to evict resources as well as which of the evictable resources to evict. Some of the common strategies used in determining which resources to evict include "Least Recently Used" (LRU) and "Least Frequently Used" (LFU).

> In CORBA, an application can use LRU or LFU strategies to allow the POA to evict servants that are not frequently used.

In addition, a user defined strategy can be used that may take different parameters in determining which resource to evict. For example, a strategy may take into account how expensive it is to re-acquire an evicted resource. Using such a strategy, resources that are less expensive to re-acquire may be evicted even if they have been more frequently used compared to resources that are more expensive to re-acquire.

4 *Define the usage of eviction in the system*: The business logic of evicting resources needs to be added to the Evictor. This includes determining how and when resources should be evicted as well as actually marking resources to be evicted. Typically, the Evictor exists as a separate object or component in the application and is configured with the necessary eviction strategy by the application. For example, an application may choose to evict resources only when available memory goes below a certain threshold. A different application, on the other hand, may implement a more proactive policy and may periodically evict resources even if memory does not go below a certain threshold.

An Evictor can use the Interceptor [POSA2] pattern to intercept user access of an object. The Interceptor can mark the object as being recently used in a manner that is completely transparent to the user. Periodically or on demand, the Evictor will typically query all evictable objects to determine which object(s) if any to evict. In the Java interface described above, the Evictor will invoke the method `info()` on each object and use the information it receives in the context of the eviction strategy to determine whether or not to evict the object.

Example Resolved Consider the example of an NMS that is responsible for managing a network of several NEs. The details of the NEs can be fetched at system start-up time or when the user makes a request for them. Without *a priori* knowledge of a user's intentions, resource usage needs

to be optimized so that only those NEs are kept in memory that are frequently accessed by the user. The eviction strategy used for this example is therefore to evict NEs that have not been accessed by the user for a threshold amount of time.

Each NE needs to implement the Eviction interface:

```
public class NE implements EvictionInterface {
     private NEComponent [] components;
     private Date lastAccess;

     public boolean isEvictable () {
        // Assume all NEs can be evicted
        return true;
     }

     public Object info () {
        // Return the date/time of last access that
        // will then be used by the Evictor to determine
        // whether or not to evict us
        return lastAccess;
     }

     public void beforeEviction () {
        // First, release all resources currently held

        // Now, call beforeEviction() on all NE components
        // to give them a chance to release necessary resources
        for (int i = 0; i < components.length; i++) {
         components[i].beforeEviction ();
        }
     }

      // ... other NE operations ...
}
```

Similarly, all first class NE components need to implement the Eviction interface so that they can recursively release any resources when they are evicted.

The Evictor can be implemented as an object that runs in its own thread of control. This allows it to periodically check if there are any NEs that have not been accessed for a threshold duration of time.

```
public class Evictor implements Runnable {
     private NE [] nes;
     public Evictor () {
        new Thread(this).start();
     }

     public void run() {
        // For simplicity, we run forever
        while(true) {
         // Sleep for configured amount of time
         try {
            Thread.sleep(pollTime);
         }
         catch(InterruptedException e) {break; }

        // Assume "threshold" contains the date/time such
        // that any NE accessed before it will be evicted

        // Go through all the NEs and see which ones to evict
        for(int i = 0; i < nes.length; i++) {
         NE ne = (NE) nes[i];
         if (ne.isEvictable())
          {
            Date d = (Date) ne.info();
            if (d.before(threshold))
             {
```

```
                    ne.beforeEviction ();
                    // Now remove the NE (application-specific)
                  }
                }
              }
            }
          }
      }
```

Note that the information that is returned by the method `info()` and how the Evictor interprets that information is application-specific and can be tailored according to the eviction strategy that needs to be deployed.

Variants

Deferred Eviction: The process of evicting one or more objects can be refined into a two-step process. Instead of removing the objects immediately, they can be first put into some kind of a FIFO queue. When the queue gets filled, the object at the head of the queue is evicted. The queue therefore serves as an intermediate holder of objects to be evicted giving the objects a "second chance." If any of these objects are accessed prior to being removed from the queue, they need not incur the cost of creation and initialization. This variant, of course, incurs the cost of maintaining a queue and also the resources associated with keeping the evicted objects in the queue.

Evictor with Object Pool: An object pool can be used to hold evicted objects. In this variant, when an object is evicted, instead of removing it from memory completely, it simply loses its identity and becomes an anonymous object. This anonymous object is then added to the object pool if it is not completely full. If the object pool is already at its maximum capacity, the object is removed from memory. When a new object needs to be created, an anonymous object from the queue can be dequeued and given a new identity. This reduces the cost of object creation. The size of the object pool should be set according to available memory.

Eviction Wrapper: An object that is evictable need not implement the Eviction interface directly. Instead, the object can be contained inside a wrapper object [POSA2] that then implements the Eviction interface. The Evictor will invoke the `beforeEviction()` method on the wrapper object that in turn is responsible for evicting the actual object. This variant makes it easier to integrate legacy code without requiring existing classes to implement the Eviction interface. An example of this variant is the use of reference objects as wrapper objects in Java. Please see the section Known Uses for further details of this example.

Known Uses

EJB—The EJB specification defines an activation and deactivation mechanism that can be used by the container to swap out beans from memory to secondary storage, thus freeing memory for other beans that need to be activated. The bean instances must implement the method `ejbPassivate()` and release any acquired resources. This method is called by the container immediately before swapping out the bean.

Java—The release of JDK 1.2 introduced the concept of *reference objects* that can be used to evict an object when heap memory is running low or when the object is no longer being used. A program can use a reference object to maintain a reference to some other object in such a way that the latter object may still be reclaimed by the garbage collector when memory is running low. In addition, the Reference Objects API defines a data structure called a *reference queue* onto which the garbage collector places the reference object prior to evicting it. An application can use the reference queue to take necessary action when certain objects become softly, weakly, or phantomly reachable and hence ready to be evicted.

CORBA—To manage memory consumption in an application, a Servant Manager typically keeps an upper bound on the total number of instantiated servants. If the number

of servants reaches a specified limit, the servant manager can evict an instantiated servant and then instantiate a servant for the current request [HeVi99].

Paging—Most operating systems that support virtual memory make use of the concept of paging or swapping. The OS copies a certain number of pages from the storage device into main memory. When a program needs a page that is not in main memory, the OS evicts a page from memory and copies it to the storage device. It then copies the required page from the storage device into main memory. This allows the OS to keep an upper bound on the total number of pages in main memory.

Consequences There are several **benefits** of using the Evictor pattern:

- *Scalability*: The Evictor pattern allows an application to keep an upper bound on the amount of resources that are being used and hence in memory at any given time. This allows an application to scale without impacting total memory consumption.
- *Low-memory Footprint*: The Evictor pattern allows an application to control via configurable strategies which resources should be kept in memory and which resources should be released. By keeping only the most essential resources in memory, an application can be kept lean as well as more efficient.
- *Transparency*: Using the Evictor pattern is completely transparent to the user and therefore does not impact application usage from the user's perspective.
- *Reliability*: The Evictor pattern reduces the probability of resource exhaustion and thus increases the stability of an application.

There are some **liabilities** of using the Evictor pattern:

- *Overhead*: The Evictor pattern requires additional business logic to determine which resources to evict and to implement the eviction strategy. In addition, the actual eviction of resources can incur a significant execution overhead.
- *Re-acquisition Penalty*: If an evicted resource is required again, the resource would need to be re-acquired. This can be expensive and can hinder application performance. The probability of this happening can be reduced by fine tuning the strategy used by the Evictor to determine which resources to evict.

See Also The Leasing design pattern [JaKi00] describes how the usage of resources can be bound by time thus allowing unused resources to be released automatically. The Lazy Acquisition design pattern [Kirch01] describes how resources can be (re-)acquired at the latest possible point in time during system run-time in order to optimize resource usage.

The Resource Exchanger design pattern [SaCa96] describes how to reduce a server's load in allocating and managing resources. While the Evictor design pattern deals with releasing resources to optimize memory management, the Resource Exchanger design pattern deals with exchanging resources to minimize resource allocation overhead.

Acknowledgements

Thanks to the *patterns team* at Siemens AG and to my shepherd, John Vlissides, for their feedback and valuable comments.

References

[BRJ98] G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

[GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[HeVi99] M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, Addison-Wesley Longman, Inc., 1999

[JaKi00] P. Jain and M. Kircher, *Leasing Pattern*, Pattern Language of Programs conference, Allerton Park, Illinois, USA, August 13-16, 2000

[KiJa00] M. Kircher and P. Jain, *Lookup Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 5-9, 2000

[Kirch01] M. Kircher, *Lazy Acquisition Pattern*, European Pattern Language of Programs conference, Kloster Irsee, Germany, July 4-8, 2001

[POSA2] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann: *Pattern-Oriented Software Architecture—Patterns for Concurrent and Distributed Objects*, John Wiley and Sons, 2000

[SaCa96] A. Sane and R. Campbell, *Resource Exchanger*, in J. Vlissides, J. Coplien, and N. Kerth (eds.), Pattern Languages of Program Design, Volume 2, Addison-Wesley, 1996